CS3025 Compiladores

Laboratorio 10.1 – 16 octubre 2023

Analisis Semantico – Type Checking II

El folder lab10.1 contiene la implementación del parser y printer del modificado lenguaje IMP definido por la siguiente sintaxis:

```
Program ::= VarDecList FunDecList
VarDecList ::= (VarDec)*
FunDecList ::= (FunDec)+
FunDec ::= ”fun” Type id “(“[ParamDecList]“)” Body “endfun”
ParamDecList ::= Type id (“,” Type id)*
VarDec ::= “var” Type VarList “;”
Type ::= id
VarList :: id (“,” id)*
StatementList ::= Stm (“;” Stm)* …
Stm ::=   id “=” CExp |
          “print” “(“ CExp “)” |
          “if” CExp “then” Body [“else” Body] “endif” |
          “while” CExp “do” Body “endwhile” |
          return “(“ [CExp] “)”
 CExp ::= …
Factor ::= … | id “( [ArgList] “)”
ArgList ::= CExp (“,” CExp)*
```

Ademas contiene la implementación incompleta, pero compilable, del verificador de tipos (typecheker) `imp_typechecker.cpp`, y un par de archivos nuevos `imp_type.hh` e `imp_type.cpp`. Compilar y probar el programa con el ejemplo `ejemplo4.imp`.

1.0 Análisis Sintactico – lo nuevo

El lenguaje modificado IMP permite la declaración y llamado a funciones. Los cambios a la gramatica generaron modificaciones a las funciones de análisis sintactico (parsing) del código en lab9.1 (`imp_parser.cpp`). Estas modificaciones permiten el analisis sintactico de:

- Listas de declaraciones de funciones FunDec.
- Sentencias de retorno de funcion de la forma `return “(“ [CExp] “)”`.
- La expresión que implementa llamados a funciones de la forma `id “( [ArgList] “)”` (function calls).

Dada la nueva clase ReturnStm(Exp* e), el análisis sintactico de sentencias de retorno de funcion se realiza de la siguiente forma:

```
Stm* Parser::parseStatement() {
  Stm* s = NULL; Exp* e = NULL;
  if (match(Token::ID)) {
      …
  } else if (match(Token::RETURN)) {
    if (!match(Token::LPAREN)) parserError("Esperaba 'lparen'");
```

```
    if (!check(Token::RPAREN))
      e = parseCExp();
    if (!match(Token::RPAREN)) parserError("Esperaba 'rparen'");
    s = new ReturnStatement(e);
      …
```

Dada la nueva clase FCallExp(string fname, list<Exp*> args), el análisis sintactico de expresiones de llamado de funcion (function calls) se implementa del siguiente modo:

```
Exp* Parser::parseFactor() {
      …
    if (match(Token::ID)) {
      string lex = previous->lexema;
      // function call
      if (match(Token::LPAREN)) {
        list<Exp*> args;
        if (!check(Token::RPAREN)) {
          args.push_back(parseCExp());
          while(match(Token::COMMA)) {
        args.push_back(parseCExp());
           }
      }
      if (!match(Token::RPAREN)) parserError("Expecting rparen");
      return new FCallExp(lex,args);
    } else // id puro
      return new IdExp(lex);
 }
   …
```

Notese que el parseo de IdExp se convierte en un caso especial de FCallExp dado que ambas expresiones empiezan con un ID.

Las declaraciones de funciones están representadas por la nueva clase

FunDec(string fname, rtype; list<string> types, vars; Body* body)

El análisis sintactico tiene la siguiente forma:

```
FunDec* Parser::parseFunDec() {
  FunDec* fd = NULL;
  if (match(Token::FUN)) {
    Body* body = NULL;
    if (!match(Token::ID)) parserError("Expecting return type …");
    string rtype = previous->lexema;
    if (!match(Token::ID)) parserError("Expecting function name …");
    string fname = previous->lexema;
    if (!match(Token::LPAREN)) parserError("Esperaba LPAREN …");
    list<string> types;
    list<string> vars;
    if (!check(Token::RPAREN)) {
      if (!match(Token::ID)) parserError("Expecting type in fun declaration");
      types.push_back(previous->lexema);
      if (!match(Token::ID)) parserError("Expecting identifier …");
      vars.push_back(previous->lexema);
      while(match(Token::COMMA)) {
      types.push_back(previous->lexema);
      if (!match(Token::ID)) parserError("Expecting identifier …");
      vars.push_back(previous->lexema);
      }
    }
    if (!match(Token::RPAREN)) parserError("Esperaba RPAREN …");
```

```
    body = parseBody();
    if (!match(Token::ENDFUN)) parserError("Esperaba ENDFUN …");
    fd = new FunDec(fname, types, vars, rtype, body);
  }
  return fd;
}
```

2.0 El TypeChecker: initializacion

La clase `ImpType` que implementa los tipos validos del lenguaje IMP tiene ahora sus propios archivos, imp_type.hh e imp_type.cpp. La interface es:

```
class ImpType {
public:
  enum TType { NOTYPE=0, VOID, INT, BOOL, FUN };
  static const char* type_names[5];
  TType ttype;
  vector<TType> types;
  bool match(const ImpType&);
  bool set_basic_type(string s);
  bool set_basic_type(TType tt);
  bool set_fun_type(list<string> slist, string s);
private:
  TType string_to_type(string s);
};
```

El verificador de tipos (typecheker) implementado por `imp_typechecker.hh/cpp` ha sido adaptado para visitar los nuevos elementos del AST. Ademas, tiene un nuevo campo `maintype` para representar el tipo de la funcion `main`, obligatoria en nuestro lenguaje, y un flag `has_main` que indica si la funcion `main` ha sido declarada o no.

Modificar el constructor de ImpTypeChecker para initializar `maintype` y `has_main`:

```
ImpTypeChecker::ImpTypeChecker():inttype(), booltype() {
  inttype.set_basic_type("int");
  booltype.set_basic_type("bool");
  voidtype.set_basic_type("void");
  // maintype y has_main
}
```

3.0 Declaraciones de funciones

Un programa IMP esta compuesto por una seria de declaraciones de variables globales seguida de declaraciones de funciones. Las reglas de IMP requieren la declaración de la funcion `main`, sin argumentos y de tipo de retorno `void`. En esta sección veremos los chequeos y acciones necesarias al nivel de declaraciones de funciones para hacer posible la verificación de tipos en el resto del programa. La mayoría de las acciones serán implementadas por las funciones `visit` en imp_typecheker.cpp.

A continuación se listan los cambios a realizarse - teniendo en cuenta que el analisis de las declaraciones de variables y la mayoría de sentencias y expresiones está incluido en el código inicial.

visit(Program *p)
Este es el punto de entrada al analisis. Previo al analisis (visit) de las declaraciones globales de variables y funciones, debemos agregar el nivel inicial a nuestro environment – este seria el ámbito global (global scope). Las acciones por ejecutar serian:

- Agregar un nivel al environment.
- Analizar (visitar) las declaraciones globales de variables y funciones.
- Verificar que la funcion main fue declarada (usar has_main).
- Remover el nivel global del environment (por simetría).

visit(FunDecList* s)
Nuestro lenguaje permite el llamado a funciones que han sido declaradas luego de su uso. Esto quiere decir que, al analizar el cuerpo de una funcion, todas las funciones (nombre y tipo) declaradas tienen que haber sido agregadas al ámbito global. Esto requiere 2 visitas a la lista de declaraciones, la primera para agregar las funciones al environment y la segunda para analizar cada declaración. Supongamos que existe la funcion `add_fundec(FunDec*)` que agrega, previa verificacion de tipos, la funcion declarada al environment global. Los pasos serian:

- Para cada fd en la lista de declaraciones, llamar a `add_fundec(fd)`.
- Para cada fd en la lista de declaraciones, analizar fd, es decir, llamar a `fd->accept(this)`.

El código de `add_fundec` debe hacer lo siguiente:

- Verificar que el tipo declarado es un tipo valido i.e. los tipos de los parámetros y tipo de retorno son válidos (esto lo hace `set_fun_dec`).
- Si la funcion es `main`, verificar que tiene el tipo correcto. Actualizar `has_main`.
- Agregar el nombre de la funcion al environment.

visit(FunDec* fd)
¿Qué tenemos? La verificacion de la correctitud del tipo de la funcion declarada ya ha sido hecho por `add_fundec`. En este paso, también se agrego el nombre de la funcion al environment. Esta información será necesaria para analizar las expresiones de llamada a funciones y la sentencia return. Para esta última, necesitamos saber el tipo de retorno de la funcion. Para esto, agregaremos al environment la nueva variable `return` para poder recobrar el tipo de retorno en el cuerpo de la funcion. Los pasos a seguir son los siguientes:

- Agregar un nivel al environment.
- Agregar al environment cada parámetro con su tipo.
- Agregar la variable `return`, con el tipo de retorno de la funcion declarada, al environment.
- Analyzar el cuerpo (body) de la funcion.
- Remover el nivel.

4.0 El resto

En esta sección indicamos los pasos necesarios para implementar el análisis de verificacion de tipos de las expresiones de llamada a funciones (function call) y las sentencias de retorno de funciones (return).

visit(ReturnStatement* s)
Necesitamos verificar que el tipo de la expresión en el `return` (void si no hay expresión) es el mismo que

el tipo de retorno de la funcion donde se encuentra el `return`. Para esto usaremos la variable return guardada en el environment. Los pasos a seguir son:

- Determinar el tipo (etype) de la expresión retornada por `return`. Si no hay expresión, este tipo debería ser `void`.
- Obtener el tipo de retorno (rtype) de la funcion donde esta el `return` (usar variable “return”).
- Verificar que los tipos coinciden.

FCallExp(string fname, list<Exp*> args)
Los pasos a seguir son:

- Verificar que fname existe y es una funcion declarada.
- Verificar que el número de argumentos es el mismo que el número de parámetros.
- Verificar que el tipo de los argumentos corresponde al tipo de los parámetros.